ELSEVIER

Information Processing Letters 69 (1999) 31–37

Information Processing Letters

Evaluations of domino-free communication-induced checkpointing protocols

Jichiang Tsai [a,1], Yi-Min Wang [b], Sy-Yen Kuo [a,*,1]

[a] *Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan*
[b] *Microsoft Research, Microsoft Corporation, Redmond, WA, USA*

Received 7 May 1998; received in revised form 23 July 1998
Communicated by D. Gries

Abstract

We give a detailed analysis of communication-induced checkpointing protocols that are free of domino effect. We investigate the validity of a common intuition in the literature and demonstrate that there is no optimal on-line domino-free protocol in terms of the number of forced checkpoints. Formal proofs on comparing existing protocols in the literature are given.

Keywords: Distributed systems; Fault tolerance; Domino effect; Communication-induced checkpointing; Rollback-recovery

1. Introduction

A distributed computation consists of a finite set of processes that communicate and synchronize with each other by exchanging messages through a network. A local checkpoint is a snapshot of the local state of a process, saved on nonvolatile storage to survive process failures. It can be reloaded into volatile memory in case of a failure to reduce the amount of lost work. When a process records such a local state, we say that this process takes a (local) checkpoint. The set of messages and the set of local checkpoints form the *checkpoint and communication pattern* associated with the distributed computation. A global checkpoint M [11] is a set of local checkpoints, one from each process; M is *consistent* if no message is sent after a checkpoint in M and received before another checkpoint in M [2].

If local checkpoints are taken independently, there is a risk that no consistent global checkpoint can ever be formed from them. This is the well-known problem of *domino effect* [10], in which unbounded, cascading rollback propagation can occur during the process of finding a consistent global checkpoint. Many protocols have been proposed to selectively take local checkpoints to eliminate the possibility of domino effect (see [3]).

One way to avoid the domino effect is to prevent any checkpoint from becoming a *useless checkpoint* [4], i.e., a checkpoint that cannot belong to any consistent global checkpoint. In this paper, we use the term *domino-free checkpointing protocol* to refer to protocols that guarantee no useless checkpoints. Coordinated checkpointing protocols [2,6] achieve domino-freedom by synchronizing the checkpointing actions of all processes through explicit control messages. In

* Corresponding author. Email: sykuo@cc.ee.ntu.edu.tw.

[1] Tsai and Kuo's work is supported by the National Science Council, Taiwan, ROC, under Grant NSC 87-2213-E-259-007.

0020-0190/99/$ – see front matter
PII: S0020-0190(98)00183-5

contrast, *communication-induced checkpointing protocols* [5] achieve coordination by piggybacking control information on application messages. Specifically, in addition to taking *basic* checkpoints independently, each process can also be asked by the protocol to take additional *forced* checkpoints so as not to make any existing checkpoint become useless. The protocol makes its decision based on the piggybacked information as well as local control variables.

Since forcing additional checkpoints incurs runtime overhead, it is desirable to force as few checkpoints as possible while still guaranteeing domino-freedom. Throughout this paper, we compare *the performance of protocols* in terms of *the number of forced checkpoints*. Intuitively, if a protocol forces checkpoints only at a *stronger* condition, then it should force fewer checkpoints overall. More specifically, if protocol A forces a checkpoint whenever condition Y is true, protocol B forces a checkpoint whenever condition Z is true, and condition Y implies condition Z, then protocol A should always outperform protocol B. In other words, it seems true that we can always improve upon a protocol by "sharpening" the condition under which it forces a checkpoint. Moreover, we can derive an optimal protocol, which always forces fewer checkpoints than any other protocol, by finding the strongest (or minimal) condition under which any protocol must force a checkpoint to avoid generating useless checkpoints.

In this paper, we construct a counterexample to prove that the above common intuition is in fact false. Basically, when comparing two protocols, the intuition is true only up to the first forced checkpoint. Beyond that, the two patterns may diverge and no longer have a common ground for comparing the conditions. For example, a protocol may unnecessarily force a checkpoint at an earlier execution point but end up outperforming other "more intelligent" protocols because it changes the pattern and happens to avoid two forced checkpoints later. Following the same arguments, we also prove that there cannot exist an optimal, domino-free communication-induced checkpointing protocol. Finally, we present a technique for formally comparing the performance of a family of domino-free protocols.

2. Preliminaries

2.1. Execution model

A distributed computation consists of a finite set P of n processes $\{P_1, P_2, \ldots, P_n\}$ that communicate and synchronize only by exchanging messages. We assume that each ordered pair of processes is connected by an asynchronous, reliable, directed logical channel with unpredictable but finite transmission delays. Processes fail according to the fail-stop model.

A process can execute *internal*, *send*, and *receive* statements. An internal statement does not involve any communication. When P_i executes "*send*(m) to P_j", it puts message m into the channel from P_i to P_j. When P_i executes "*receive*(m)", it is blocked until at least one message directed to P_i has arrived. Then, a message is withdrawn from one of its input channels and delivered to P_i. Executions of internal, send, and receive statements are modeled by internal, sending, and receiving events, respectively [4].

Processes of a distributed computation are *sequential*: each process P_i produces a *sequence* of events. All the events produced by a distributed computation can be modeled as a partially ordered set with Lamport's well-known *happened-before* relation "$\overset{hb}{\rightarrow}$", defined as follows [7].

Definition 1. The relation "$\overset{hb}{\rightarrow}$" on the set of events satisfies the following three conditions:

(1) If a and b are events in the same process and a comes before b, then $a \overset{hb}{\rightarrow} b$.
(2) If a is the event *send*(m) and b is the event *receive*(m), then $a \overset{hb}{\rightarrow} b$.
(3) If $a \overset{hb}{\rightarrow} b$ and $b \overset{hb}{\rightarrow} c$ then $a \overset{hb}{\rightarrow} c$.

Given a distributed computation H, its associated checkpoint and communication pattern is the set of local checkpoints in H partially ordered by the set of messages. Fig. 1 shows an example checkpoint and communication pattern. $C_{i,x}$ represents checkpoint x of process P_i where i is the *process id* and x the *checkpoint index*. The sequence of events occurring at P_i between $C_{i,x-1}$ and $C_{i,x}$ $(x > 0)$, called a *checkpoint interval*, is denoted by $I_{i,x}$. We assume that each process P_i starts its execution with an initial checkpoint $C_{i,0}$.

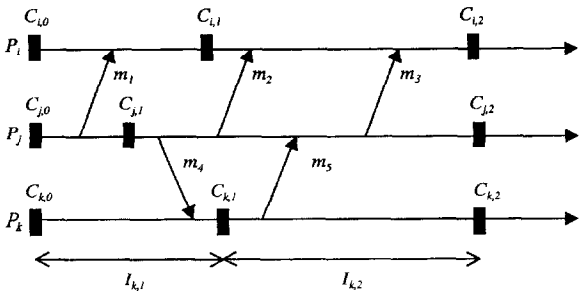

Fig. 1. A checkpoint and communication pattern.

2.2. Z-cycles and useless checkpoints

Netzer and Xu introduced the following notion of *Z-paths* and *Z-cycles* and proved that a checkpoint $C_{i,x}$ is useless *iff* it is involved in a Z-cycle.

Definition 2. A *Z-path* is a sequence of messages $[m_1, m_2, \ldots, m_q]$ $(q \geqslant 1)$ such that, for each i, $1 \leqslant i \leqslant q-1$:

$receive(m_i) \in I_{k,s} \wedge send(m_{i+1}) \in I_{k,t} \wedge s \leqslant t$ [9].

If a Z-path $[m_1, m_2, \ldots, m_q]$ satisfies the condition that $send(m_1) \in I_{i,x+1}$ ($C_{i,x}$ precedes $send(m_1)$) and $receive(m_q) \in I_{j,y}$ ($receive(m_q)$ precedes $C_{j,y}$), we say that this Z-path is from $C_{i,x}$ to $C_{j,y}$. A Z-path from a local checkpoint $C_{i,x}$ to the same local checkpoint $C_{i,x}$ is called a *Z-cycle*. We say that it involves the local checkpoint $C_{i,x}$.

Definition 3. A local checkpoint is *useless* [4] *iff* it does not belong to any consistent global checkpoint.

Theorem 4. *A local checkpoint $C_{i,x}$ is useless iff it is involved in a Z-cycle* [9].

For example, in Fig. 1, both the message sequences $[m_5, m_2]$ and $[m_5, m_3]$ constitute Z-paths from $C_{k,1}$ to $C_{i,2}$. According to Theorem 4, $C_{k,1}$ is useless because it is involved in the Z-cycle $[m_5, m_4]$. It becomes clear that any domino-free checkpointing protocol (as defined in the previous section) must eliminate all Z-cycles by forcing additional checkpoints. (We say that all Z-cycles are *broken*.)

One approach to avoiding generating any Z-cycles is to have a checkpoint timestamp algorithm that guarantees that *timestamps for checkpoints always increase along any Z-path* [4,8], as stated in the following theorem:

Theorem 5. *Let each checkpoint C be associated with a timestamp $C.t$. If for every pair of checkpoints $C_{i,x}$ and $C_{j,y}$ with a Z-path from $C_{i,x}$ to $C_{j,y}$ we have $C_{i,x}.t < C_{j,y}.t$, then there is no Z-cycle.*

The timestamps can be maintained in the following classical way [7]:

- each process P_i manages a local logical clock lc_i;
- before taking a (basic or forced) checkpoint, P_i increments lc_i by Theorem 1 and assigns the new value to be the timestamp of the checkpoint;
- upon sending a message m, P_i timestamps m with its current lc_i (let $m.t$ denote the timestamp of m);
- upon receiving a message m, P_i updates lc_i to $\max(lc_i, m.t)$.

See the Z-path depicted in Fig. 2(a), which contains only two messages m_1 and m_2 with $send(m_2) \overset{hb}{\rightarrow} receive(m_1)$ and is from one process to a different process. We call this type of Z-path an *MM-path*. In contrast, the Z-paths shown in Figs 2(b) and (c), which also contain two messages m_1 and m_2 with $send(m_2) \overset{hb}{\rightarrow} receive(m_1)$ but are from a process to itself, are called *MM-rings*. To ensure monotonically increasing checkpoint timestamps, we consider the following four cases:

- MM-paths with $m_1.t \leqslant m_2.t$: Since the timestamps increase along the path, P_j does not need to force a checkpoint.
- MM-paths with $m_1.t > m_2.t$: Since the timestamps may not increase along the path, a safe strategy to prevent Z-cycles is for P_j to take a forced checkpoint before delivering m_1 so that the sequence $[m_1, m_2]$ is no longer a Z-path.
- MM-rings not involving checkpoints (unbreakable MM-rings): An MM-ring from a checkpoint to a later checkpoint of the same process is called an *unbreakable* MM-ring. For example, Fig. 2(b) shows an unbreakable MM-ring from P_k to itself. Since the later checkpoint must have a larger timestamp value, an unbreakable MM-ring trivially satisfies the assumption of Theorem 5 and does not require any forced checkpoint to break the ring.
- MM-rings involving checkpoints (breakable MM-rings): In contrast, a breakable MM-ring is from

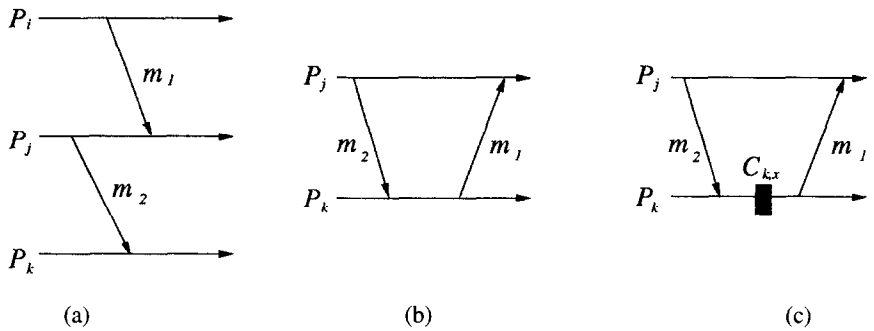

Fig. 2. (a) An MM-path; (b) an unbreakable MM-ring; (c) a breakable MM-ring.

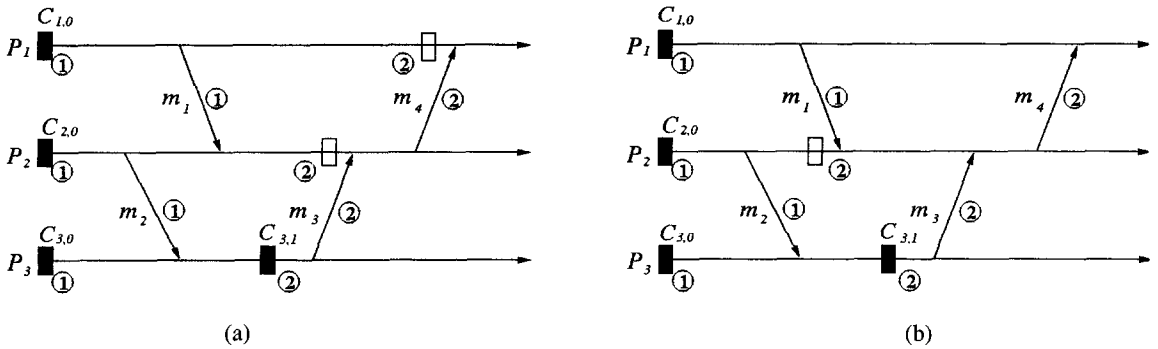

Fig. 3. The scenario of the counterexample (a) CPn; (b) CPm.

a checkpoint to itself or an earlier checkpoint of the same process. For example, Fig. 2(c) shows a breakable MM-ring from $C_{k,x}$ to itself. Since it is impossible for a breakable MM-ring to satisfy the assumption of Theorem 5, P_j must force a checkpoint to break the ring.

Note that it has been shown in [4] that one integer vector and one Boolean vector are sufficient for distinguishing whether an MM-ring involves any local checkpoint or not. A protocol that detects these two kinds of rings is described in [4]. Moreover, [4] describes a family of timestamp-based checkpointing protocols, whose performance will be formally compared in a later section.

3. Common intuition is false

It seems natural to try to improve the performance of protocols by piggybacking more causal information on each message and sharpening the conditions for forcing checkpoints, like the method used in [4]. Intuitively, if protocols A and B force checkpoints under conditions Y and Z, respectively, and Y implies Z, then protocol A should perform at least as well as protocol B. In this section, we use a counterexample to show that the above common intuition is false. The main reason is that forced checkpoint may change the subsequent pattern, and it may no longer be meaningful to consider the relation "Y implies Z" across two different patterns.

Counterexample. Let CPn be a protocol that breaks any MM-paths $[m_1, m_2]$ with $m_1.t > m_2.t$ and all breakable MM-rings, and CPm be a protocol that breaks every MM-path $[m_1, m_2]$ with $m_1.t \geqslant m_2.t$ and all breakable MM-rings. Clearly, both protocols achieve monotonically increasing checkpoint timestamps along all Z-paths, and so are domino-free. Also, $m_1.t > m_2.t$ implies $m_1.t \geqslant m_2.t$.

Figs 3(a) and (b) show the execution results of CPn and CPm, respectively, on the same checkpoint and

communication pattern. Integers in circles indicate the timestamps of checkpoints and messages. Black rectangular boxes represent basic checkpoints, and hollow boxes represent forced checkpoints. As shown in Fig. 3(a), P_2 forces a checkpoint before delivering m_3 to break the breakable MM-ring, and that checkpoint in turn forces P_1 to take another checkpoint before delivering m_4. In contrast, in Fig. 3(b), CPm needs only one forced checkpoint to make the same pattern domino-free. This counterexample shows that CPm forces fewer checkpoints than CPn in the given checkpoint and communication pattern, although it forces checkpoints at a weaker condition. This serves to disprove the common intuition.

In the next section, we will use similar techniques to prove that there is no optimal on-line domino-free protocol. This scenario is quite common in the area of on-line algorithms due to the lack of knowledge of the future.

4. No optimal on-line protocol

We base our proof on the pattern used in the previous counterexample, which is redrawn in Fig. 4 and denoted as *ccpat*. Because the MM-ring $[m_3, m_2]$ is breakable, we immediately have the following lemma:

Lemma 6. *Any domino-free protocol must force process P_2 in ccpat to take at least one checkpoint between points a and b.*

Lemma 7. *Any domino-free protocol that forces process P_2 in ccpat to take a checkpoint between points c and b, must also force process P_1 to take another forced checkpoint.*

Proof. If P_2 takes a checkpoint between c and b, then MM-ring $[m_4, m_1]$ becomes breakable. As a result, P_1 has to take another checkpoint to break this MM-ring in order to make the pattern domino-free. □

Theorem 8. *There is no optimal on-line domino-free protocol.*

Proof. Consider the checkpoint and communication pattern *ccpat* in Fig. 4. Since the domino-free protocol

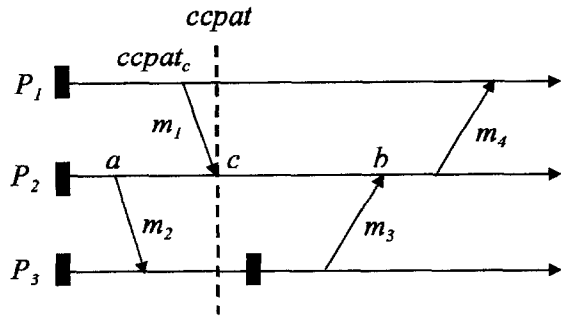

Fig. 4. Patterns *ccpat* and $ccpat_c$.

CPm in the counterexample illustrated previously needs to take only one forced checkpoint, protocols that force any checkpoint between c and b need at least two forced checkpoints by Lemma 7 and therefore cannot be optimal. Trivially, protocols that take more than one checkpoint between a and c cannot be optimal either. So an optimal domino-free protocol, if any, must take exactly one forced checkpoint between a and c according to Lemma 6.

Now suppose there exists an optimal *on-line* domino-free protocol CPo, which must take exactly one forced checkpoint between a and c. Consider the pattern $ccpat_c$ in Fig. 4, shown as the portion to the left of the dotted line. Since $ccpat_c$ and *ccpat* have exactly the same execution history up to the dotted line, the on-line protocol CPo must still take exactly one forced checkpoint between a and c. Similarly, the on-line protocol CPn in the counterexample still takes zero forced checkpoint in $ccpat_c$, contradicting the optimality of CPo. □

Since on-line protocols CPn and CPm are based only on causal history, as opposed to omniscient knowledge of the entire execution history, the same proof holds for the following corollary:

Corollary 9. *There is no optimal on-line domino-free protocol based on causal history.*

5. Techniques for protocol performance comparison

Results from the previous sections indicate that the comparison of domino-free protocol performance can-

not be based on that of checkpoint-inducing conditions. Hence we demonstrate in this section formal approaches to comparing several existing protocols. The underlying condition of each protocol proposed in the literature can be expressed as $(m.t > lc_i) \wedge P$ where P is a predicate characterizing the protocol [4]. The MS protocol proposed in [8], which is a variation of the protocol in [1], is exactly based on the "$(m.t > lc_i)$" condition; that is, it directs process P_i to force a checkpoint when receiving a message m with a timestamp larger than the local clock of P_i. Two other protocols are proposed in [4]: protocol HMNR1 forces a checkpoint when encountering an "$m.t > lc_i$" condition after a sending event in the same interval, while protocol HMNR2 is more restrictive and takes a forced checkpoint only for some particular "$m.t > lc_i$" conditions after a sending event in the same interval. Both MS and HMNR1 are simplified versions of HMNR2 [4]. Simulation results comparing these three protocols have been presented in [4]. We now describe an approach to formally comparing their performance by demonstrating that the common intuition is in fact valid for this particular family of protocols, denoted by F_E.

Lemma 10. *Given any checkpoint and communication pattern, the timestamps produced by different protocols in the F_E family are equivalent at every execution point.*

Proof. For F_E protocols, a process may update its local clock according to the management of timestamps only when a basic checkpoint is taken or an "$m.t > lc_i$" condition is encountered. Because all F_E protocols take the same basic checkpoints, it is sufficient to consider "$m.t > lc_i$" conditions. Since no forced checkpoint can be taken before the first "$m.t > lc_i$" condition, all F_E protocols will encounter the first "$m.t > lc_i$" condition at the same execution point. Now, no matter a forced checkpoint is taken or not, all F_E protocols will update their local clocks to the same value of $m.t$ in the max() operation. By repeating the same argument for all subsequent "$m.t > lc_i$" conditions, this lemma is proved. □

It directly follows from the previous lemma that HMNR1 outperforms MS. This is because, whenever HMNR1 forces a checkpoint, the "$m.t > lc_i$" condition must also be true in the MS case to force a checkpoint at the same execution point. We next show that any protocol CPs based on a stronger condition than HMNR1's outperforms HMNR1. Let C_h and C_s represent the checkpoint-inducing conditions of HMNR1 and CPs, respectively. By definition, we have $C_s \Rightarrow C_h$.

Lemma 11. HMNR1 *must force at least one checkpoint between any two consecutive forced checkpoints taken by* CPs *in a process.*

Proof. First, consider the pattern produced by CPs. Given any two consecutive forced checkpoints, consider the second one and call it c_2. Since $C_s \Rightarrow C_h$, there exists at least one sending event between c_2 and its nearest previous checkpoint (basic or forced). Let e denote one such event that is closest to c_2. Clearly, e is between the two consecutive forced checkpoints. Also, there is an "$m.t > lc_i$" condition at the execution point a where c_2 is taken.

Now consider the pattern produced by HMNR1, which contains the same event e. From Lemma 10, the above "$m.t > lc_i$" condition is still true at a, and we distinguish two cases: either at least one forced checkpoint has been taken between e and a or none has been taken. In the latter case, C_h is true at a, so HMNR1 must force a checkpoint at a. So, in either case, at least one forced checkpoint is taken by HMNR1 between the two consecutive checkpoints forced by CPs. □

We can then derive the following *monotonicity* property.

Lemma 12. *In any process,* HMNR1 *takes the nth forced checkpoint no later than* CPs, *for all n.*

Proof. We give a proof by induction. Since $C_s \Rightarrow C_h$, HMNR1 takes the first forced checkpoint no later than CPs. Now suppose HMNR1 takes the kth forced checkpoint no later than CPs. From Lemma 11, HMNR1 must force at least one checkpoint between the kth and the $(k+1)$th forced checkpoints of CPs, and this checkpoint must be HMNR1's mth checkpoint where $m \geqslant k+1$. Therefore HMNR1 must take its $(k+1)$th forced checkpoint no later than CPs. □

Finally, we have the following theorem.

Theorem 13. *Any domino-free protocol* CPs *that forces a checkpoint at a stronger condition than* HMNR1*'s must always take at most as many forced checkpoints as* HMNR1.

Proof. Given any checkpoint and communication pattern, CPs forces every process to take at most as many forced checkpoints as in the HMNR1 case, according to Lemma 12. Therefore, in the overall system, CPs forces at most as many checkpoints. □

Since HMNR2 belongs to the family of CPs, it directly follows from Theorem 13 that HMNR2 forces at most as many checkpoints as HMNR1.

References

[1] D. Briatico, A. Ciufoletti, L. Simoncini, A distributed domino-effect free recovery algorithm, in: Proc. 4th IEEE Symp. on Reliability in Distributed Software and Database Systems, October 1984, pp. 207–215.

[2] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75.

[3] E.N. Elnozahy, D.B. Johnson, Y.M. Wang, A survey of rollback-recovery protocols in message-passing systems, TR, CMU-CS-96-181, Carnegie-Mellon University, Pittsburgh, PA, 1996.

[4] J.M. Helary, A. Mostefaoui, R.H.B. Netzer, M. Raynal, Communication-based prevention of useless checkpoints in distributed computations, in: Proc. 16th IEEE Symp. Reliable Distributed Systems, October 1997, pp. 183–190.

[5] B. Janssens, W.K. Fuchs, Experimental evaluation of multiprocessor cache-based error recovery, in: Proc. Internat. Conf. Parallel Processing 1 (1991) 505–508.

[6] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed systems, IEEE Trans. Software Eng. 13 (1) (1987) 23–31.

[7] L. Lamport, Time, clocks and the ordering of events in a distributed system, Comm. ACM 21 (7) (1978) 558–565.

[8] D. Manivannan, M. Singhal, A low overhead recovery technique using quasi-synchronous checkpointing, in: Proc. 16th IEEE Internat. Conf. Distributed Computing Systems, May 1996, pp. 100–107.

[9] R.H.B. Netzer, J. Xu, Necessary and sufficient conditions for consistent global snapshots, IEEE Trans. Parallel Distrib. Syst. 6 (2) (1995) 165–169.

[10] B. Randell, System structures for software fault-tolerance, IEEE Trans. Software Eng. 1 (2) (1975) 220–232.

[11] Y.M. Wang, A. Lowry, W.K. Fuchs, Consistent global checkpoints based on direct dependency tracking, Inform. Process. Lett. 50 (4) (1994) 223–230.